

SEMINAR 4

1 Moștenire

Suntem deseori în situația de a avea nevoie de a avea mai multe tipuri de date care au în comun mai multe caracteristici sau, o dată cu dezvoltarea aplicației noastre, să fim nevoiți să extindem un tip de date existent fără să pierdem forma curentă. O soluție evidentă (non-POO) ar fi să cream noul tip plecând de la o copie a tipului curent. Genul acesta de soluție nu este foarte practică. Duplicarea codului, pe lângă faptul că va crește numărul de linii de cod din proiect, duce la o durată mai mare a procesului de compilare. Mai mult o problema identificată în codul comun trebuie fixată separat în fiecare copie a codului.

În programarea orientată pe obiecte putem crea tipuri noi de date plecând de la un tip de date, fără a fi nevoie să duplicăm cod. Această funcționalitate se numește *moștenire*. Astfel putem crea funcționalități noi fără să pierdem funcționalități vechi și fără să creștem dimensiunea proiectului într-o manieră necontrolată (refolosim cod).

În C++ (dar nu numai) putem moșteni unul sau mai multe tipuri de date în crearea tipului nou de date. Clasa din care se moștenește se numește clasă de bază, iar clasa care moștenește se numește clasă derivată. Sintaxă moștenire:

```
1 class <nume> : <spec_acces_1> <baza_1>, ... , <spec_acces_n> <baza_n> {  
2     // definitie clasa  
3 };
```

Specificatorii de access au rolul de specifica ce tip de access vor avea proprietățile moștenite în clasa derivată. Următorul tabel descrie cum specificatorul de access folosit la moștenire influențează specificatorul final de acces al unei proprietăți moștenite:

<div>Moștenire \ bază</div>	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	inaccesibil	inaccesibil	inaccesibil

Când instanțiem un obiect al unei clase derivate, înainte de a se executa constructorul clasei, se va apela implicit, de către compilator, constructorul fără parametrii al clasei de bază. Acest comportament poate fi modificat prin folosirea listei de inițializare în constructorul clasei derivate și apelarea explicită a constructorului clasei de bază dorit.

Când este distrus un obiect, destructorii sunt apelați în ordine inversă constructorilor: mai întâi destructorul clase de derivate, apoi constructorul clase de bază.

```
1 #include <iostream>  
2  
3 using namespace std;  
4  
5 class A {  
6     public:  
7     A () {cout << "A";}  
8     A (int x) {cout << "A" << x;}  
9     ~A () {cout << "~A";}  
10 };  
11
```

```

12 class B: public A {
13     public:
14     B () : A(3) {cout << "B";}
15     ~B () {cout << "~B";}
16 };
17
18 class C: public B {
19     public:
20     C () {cout << "C";}
21     ~C () {cout << "~C";}
22 };
23
24 int main () {
25     C c;
26     return 0;
27 }
28
29 // afiseaza A3BC~C~B~A

```

Cămpurile și metodele statice sunt și moștenite în clasa derivată. Cămpurile statice sunt partajate cu clasa de bază: o modificare a unui câmp static în clasa de bază va fi văzută și în clasa derivată și viceversa.

Dacă implementarea metodelor moștenite nu este potrivită pentru clasa derivată, atunci putem să suprascrim metoda și să îi oferim o nouă implementare.

Atunci când lucrăm cu pointeri și referințe către tipul de bază, putem asigna în acestea obiecte de tip derivat. Acest procedeu se numește upcasting și este posibil deoarece clasa derivat o parte comună cu clasa de bază. Aceeași afirmație nu este adevărată și în sens invers deoarece nu întotdeauna tipul de bază conține exact aceleași proprietăți ca tipul derivat (există downcasting însă nu e mereu posibil).

```

1 #include <iostream>
2
3 using namespace std;
4
5 class A {
6     public:
7     static void f () {
8         cout << "A" << endl;
9     }
10 };
11
12 int A::a = 0;
13
14 class B: public A {
15     public:
16     static void f () {
17         cout << "B" << endl;
18     }
19 };
20
21
22 int main () {
23     A *a = new B();
24     a->f(); // se apeleaza A::f()
25     return 0;
26 }

```

Dacă moștenim multiple clase, este posibil să avem două proprietăți cu același nume. Pentru a putea accesa proprietățile e nevoie să specificăm numele clasei de bază din care vrem să accesăm proprietatea (dacă nu facem asta compilatorul va spune că simbolul accesat este ambiguu):

```

1 #include <iostream>
2
3 using namespace std;
4
5 class A {

```

```

6     protected:
7         int x;
8 };
9
10 class B {
11     protected:
12         int x;
13 };
14
15 class C: public A, public B {
16     public:
17         C () {
18             A::x = 3;
19         }
20         friend void showC (C c);
21 };
22
23 void showC (C c) {
24     cout << c.A::x << " ";
25 }
26
27 int main () {
28     C c;
29     showC(c);
30     return 0;
31 }

```

2 Upcasting

În programarea orientată pe obiecte putem asigna un obiect de tip derivat către o variabilă / referința / pointer de tip bază. Acest fenomen se numește *upcasting*. Acest lucru este permis deoarece clasa derivată are aceleași proprietăți și metode cu clasa de bază, prin urmare un obiect de tip derivat poate substitui oricând un obiect de tip bază.

```

1 #include <iostream>
2 class B {
3     public:
4         B () {std::cout << "B";}
5 };
6
7 class D : public B {
8     public:
9         D () {std::cout << "D";}
10 };
11
12 int main () {
13     D d;
14     B b = d;
15     B *pb = &d; // B *pb = new D();
16     B &rb = d;
17     return 0;
18 }

```

Atenție! Operația inversă (asignare obiect de tip bază către variabilă / referința / pointer de tip derivat – *downcasting*) nu este întotdeauna posibilă.

3 Metode virtuale

Metodele virtuale sunt metode ale căror funcționalitate poate fi rescrisă în clasa derivată. Spre deosebire de metodele normale, funcționalitate rescrisă se păstrează atunci când un obiect de tip derivat este utilizat folosind referințe și pointeri de tip bază.

Pentru a declara o metodă virtuală trebuie adăugat cuvântul cheie **virtual** înainte de de semnatura metodei.

```

1 #include<iostream>
2 using namespace std;
3 class B {
4     public:
5     virtual void foo() {
6         cout << "B";
7     }
8 };
9 class D: public B {
10 public:
11     void foo () {
12         cout << "D";
13     }
14 };
15
16 int main() public
17 {
18     B b = D();
19     b.foo();           // B
20     B *pb = new D();
21     pb->foo();         // D
22     return 0;
23 }

```

Atunci când facem upcasting, la apelarea unei metode virtuale se folosește cea mai recentă redefinire a metodei (nu suntem nevoiți să redefinim la fiecare moștenire);

```

1 #include<iostream>
2 using namespace std;
3 class B {
4     public:
5     virtual void foo() {
6         cout << "B::foo" << endl;
7     }
8     virtual void bar() {
9         cout << "B::bar" << endl;
10    }
11 };
12 class D: public B {
13 public:
14     void foo () {
15         cout << "D::foo" << endl;
16     }
17
18     void bar() {
19         cout << "D::bar" << endl;
20     }
21 };
22
23 class E: public D {
24 public:
25     void bar () {
26         cout << "E::bar" << endl;
27     }
28 };
29
30 int main()
31 {
32     B *b = new E();
33     b->foo();           // D::foo
34     b->bar();           // E::bar
35     return 0;
36 }

```

Apelarea corectă a funcțiilor virtuale se realizează prin ***vptr** și **VTABLE**. Când declarăm o metodă virtuală, fiecare obiect al clasei are un nou membru ***vptr** pointer către tabela dinamică **VTABLE** care conține pointeri către implementările funcțiilor virtuale.

Despre metode virtuale:

- e recomandat să fie declarate public;
- metodele virtuale nu pot fi statice;
- pentru a putea folosi polimorfismul trebuie, e recomandat să fie apelate folosind pointeri sau referințe.
- Putem avea și destructori virtuali (ajută la distrugerea obiectelor asignate prin upcasting);

4 Clase abstracte

Atunci când declarăm o metodă virtuală nu suntem obligați să furnizăm mereu o implementare. Putem transforma metoda virtuală într-o *metodă virtuală pură* adăugând `= 0` după semnatura metodei.

```
1 class C {
2 public:
3     virtual void foo() = 0;
4 };
```

Putem oferi implementări pentru metodele virtuale pure, iar singura metodă prin care acestea pot fi folosite sunt prin apelarea explicită în clase derivate.

```
1 #include<iostream>
2 using namespace std;
3 class B {
4     public:
5     virtual void foo() = 0;
6 };
7
8 void B::foo() {
9     cout << "foo";
10 }
11
12
13 class D: public B {
14 public:
15     void foo () { // eliminarea acestei redefiniri
16         B::foo();
17     }
18 };
```

O clasă cu o metodă virtuală pură se numește *clasă abstractă*. O clasă abstractă nu poate fi instanțiată. Se pot declara doar pointeri și referințe către o astfel de clasă. Clasele care moștenesc o clasă abstractă trebuie să implementeze toate metodele virtuale pure.

Exerciții

1. Spuneți care dintre următoarele secvențe de cod compilează și care nu. În cazul secvențelor de cod care compilează spuneți care este outputul programului. În cazul secvențelor care nu compilează sugerați o modificare prin care secvența compilează și spuneți care este outputul secvenței modificate.

```
1 // 1
2 #include<iostream>
3 using namespace std;
4 class A {
5     public:
6     A(int x){cout<<"A"<<x;}
7     ~A(){cout<<"~A";}
8 };
9
```

```

10 class B: public A {
11 public:
12     B(int y=3){cout<<"B"<<4;}
13 };
14
15 class C: public B {
16     C():B(10) {cout<<"C";}
17 }
18
19 int main () {
20     A *pa = new C();
21     delete pa;
22     return 0;
23 }

```

```

1 // 2
2 #include<iostream>
3 using namespace std;
4
5 class A {
6     int y;
7 public:
8     A(int x = 2020) : y(x) {cout<<"A";}
9     ~A(){cout<<"~A";}
10    ostream& operator<<(ostream& out) const {
11        out<<this->y; return out;
12    }
13 };
14
15 int main () {
16     A a(2), b;
17     a << (b << cout);
18     return 0;
19 }

```

```

1 // 3
2 #include <iostream>
3 using namespace std;
4
5 class C {
6     int * const p;
7 public:
8     C(int x) : p(&x) {(*p)+=3;}
9     void set (int x) {p = &x;}
10    friend ostream& operator<<(ostream& o, C x) {
11        o << *x.p; return o;
12    }
13 };
14
15 int main () {
16     cout << C(3);
17     return 0;
18 }

```

```

1 // 4
2 #include <iostream>
3 using namespace std;
4
5 #define ltlr <<
6
7 class B {
8 public:
9     virtual void f() = 0;
10 };
11
12 void B::f(){cout ltlr "I'm so B";}
13
14 class C : public B {

```

```

15 };
16
17 int main () {
18     B* pb = new C();
19     return 0;
20 }

1 // 5
2 #include <iostream>
3 using namespace std;
4
5 class A {
6 public:
7     A () {cout<<"A";}
8     virtual ~A() = 0;
9 };
10 A::~~A() {cout<<"~A";}
11
12 class B : public A {
13 public:
14     B() {cout<<"B";}
15     ~B() {cout<<"~B";}
16 };
17
18 class C: public B {
19 public:
20     C () {cout << "C";}
21     ~C () {cout << "~C";}
22 };
23
24 int main () {
25     C c;
26     B &pb = c;
27     return 0;
28 }

```

2. Implementați o aplicație de gestionare a evenimentelor. Fiecare eveniment are un nume, o locație și o dată la care are loc (minut, ora zi, luna, an) și o durată. Evenimentele pot avea loc o singură dată sau pot fi recurente (zilnic, săptămânal, lunar, anual) și în funcție de numărul de persoane implicat, evenimentele pot fi individuale sau de grup (o listă de persoane implicate în eveniment este oferită). Fiecare eveniment are un ID unic de forma $EV - X - nr_ord$, unde nr_ord este numărul de ordine al evenimentului, iar X are valoare I pentru evenimente individuale, R pentru evenimente recurente, IR pentru evenimente individuale recurente și GR pentru evenimente de grup recurente. Fiecare eveniment are un cost de administrare:

- un eveniment individual costă 100 lei pe ora;
- un eveniment recurent primește un discount în funcție de recurența acestuia: 18% pt evenimente zilnice, 13% pt evenimente săptămânale, 7% pentru evenimente lunare 5% pentru evenimente anuale.
- un eveniment de grup costă 30 lei pe oră de persoană.

Aplicația trebuie să permită crearea de noi evenimente, listarea evenimentelor și stergerea unui eveniment după ID.